Well-formedness and typing rules for UML Composite Structures

Iulia Dragomir and Iulian Ober
IRIT - University of Toulouse
118 Route de Narbonne, 31062 Toulouse, France
{iulia.dragomir, iulian.ober}@irit.fr

Abstract

Starting from version 2.0, UML introduced hierarchical composite structures, which are an expressive way of defining complex software architectures, but which have a very loosely defined semantics in the standard. In this paper we propose a set of consistency rules that disambiguate the meaning of UML composite structures. Our primary goal was to have an operational model of composite structures for the OMEGA UML profile, an executable profile dedicated to the formal specification and validation of real-time systems, developed in a past project to which we contributed. However, the rules and principles stated here are applicable to other hierarchical component models based on the same concepts, such as SysML. The presented ruleset is supported by an OCL formalization which is described in this report. This formalization was applied on different complex models for the evaluation and validation of the proposed principles.

1 Introduction

This technical report[1] introduces a new version of the OMEGA UML Profile and its formalization, an extension based on composite structures. Composite structures were introduced starting with the version 2.0 in the UML standard [12] and represent a big evolution in the representation of complex hierarchical systems. Because the UML standard is underspecified in order to preserve the generality of the language, various ambiguities are introduced in the model when using composite structures. Our purpose is to define an expressive set of notions and principles to clarify the composite structures at the modelling level and also at the execution model level. Our rule set can be applied to other component based systems, like SysML [10] or MARTE [11]. All the principles were formalized in OCL in order to catch the most frequent modelling issues regarding composite structures.

[1]This report represents an excerpt of Iulia Dragomir's Master Thesis defended June 2010 at Université Paul Sabatier Toulouse III, France.

Our interest in composite structures is given by the powerful expressiveness of these constructs when modelling the architecture of hierarchical systems. Common applications are real-time embedded systems which can be found in a large number of domains like avionics, aeronautics, consumer electronics and many others. An important research topic is to prove their safety.

The context of our work is the previous OMEGA UML Profile, dedicated to the specification and validation of real-time embedded systems. This profile is based on a subset of UML 1.4 elements for modelling the structure and the behaviour of a system and has as extensions time modelling and observers (elements that express safety properties of a model). The profile is integrated in a platform, the IFx Toolset, which proposes validation techniques like model-checking, simulation and static analysis via a translation to the intermediate IF representation.

Related work. The idea of UML composite structures is rooted in previously existing languages, notably ROOM [21] and SDL [15]. However, UML adds much complexity with respect to previous models, e.g., by allowing explicit port behaviour specifications, multiple interfaces per port, typing of connectors with associations, etc. Many problems identified in this paper stem from the added complexity. Potential problems and ambiguities in UML composite structures have previously been discussed by other authors [20, 5]. In [5], Cuccuru et al. proposed a set of additional rules meant to further clarify the semantics of UML composite structures. While we fully subscribe to the solutions they propose, some issues remain unsolved, and the present paper is complementary to their solutions.

Structure of the report. Section 2 presents an overview of the previous version of the OMEGA Profile. Section 3 introduces the Composite Structures and the well-formedness and typing rules for these structures are presented in Section 4. Section 5 presents the principles for the model transformation to an IF model. Section 6 contains the OCL formalization for profile's rule set and Section 7 describes the evaluation of the formalization on a complex model before concluding.

2 Overview of the OMEGA UML Profile

The OMEGA Profile ([18]) identifies a subset of the UML language which is sufficiently expressive for modeling the structure and behavior of real-time systems, and for which an operational semantics is defined by closing the relevant semantic variation points left open in the UML standard ([6]). This profile it is integrated within a framework (the IFx toolset [2]) that supports techniques like static analysis, model checking and simulation for validating real-time embedded system.

The previous version of the OMEGA UML Profile is based on a subset of UML 1.4. From structural point of view the profile consists in :

- **Classes.** They can be *active* or *passive*, partitioning the object space in *activity*

groups. Each instance of an *active* class[2] defines an *activity group*. Each instance of a *passive* class belongs to one single *activity group*, the one that has created it. They can own attributes, relationships, operations and state machines. Activity groups are considered concurrent and they react to external stimuli (like signals and operation calls) in a run-to-completion manner. When a request is received from the outside environment, it is stored in group's queue and is handled later when the group is stable. By *stable* we mean that every object owned by the group has no spontaneous transitions (transitions that are guarded by a boolean condition and have no trigger) or pending operations from inside the group (i.e., the *object* is *stable*).

- **Structural features.** Classes have attributes which can have *predefined types*: Integer, Real, Boolean, or *reference types*. Since the OMEGA UML Profile was developed for modelling real-time embedded systems, two extensions representing time (*timer* and *clocks*) have been included in the profile. *Timer* objects measure durations. They may be set to a relative deadline and can be reset. Upon deadline different operations may be executed by objects: sending a signal, calling an operation, etc. *Clock* objects measure also durations, but their values can be consulted by other objects.

- **Relationships.** The relationships that can be defined between classes are *associations* and *generalisations*. The associations supported are *simple* or *compositional*.

From behavioural point of view, the OMEGA UML contains:

- **Operations.** We distinguish two types of operations: *primitive* and *triggered*. *Triggered operations* are a special kind of transition trigger: the call of such an operation enables the transition which this guards. *Primitive operations* are similar to the methods in object oriented programming languages: they are subject to polymorphism and dynamic binding because of the inheritance relationship that may be defined between classes. They can own a body which is described by an action. When an operation is called by an object from the same activity group, the call is handled immediately by the object called using a call stack. If the call is made from another activity group, then it is queued by the receiving group and handled in a later run-to-completion step.

- **Signals.** They are the second method for asynchronous communication between objects and are usually used for triggering actions in the state machine of the target object. They can have parameters and they differ from triggered operation in the sense that they cannot have a return value. Signals always pass through object's activity group and are handled in a later run-to-completion step, no matter if the target is in the same activity group as the sender or not.

- **State machines.** They describe the behaviour of a class in term of states, transitions, triggers, actions, etc.

[2]Active classes are represented with a thick border to distinguish them for passive classes.

- **Actions.** They describe the effect of a transition in a state machine or the body of an operation. The OMEGA UML Profile introduces a textual action language, OMAL, compatible with the action metamodel of UML and which covers notions like: object creation/destruction, operation calls, expression evaluation, variable assignments, signal output, return action and control flow structuring statements (if-then-else and do-while).

Besides the timing extension, OMEGA UML Profile introduces the notion of *UML Observers*. They are special objects that monitor the system, respectively its states and its events. Observers are modelled by classes stereotyped with `<<observer>>`. They have local memory and a state machine describes their behaviour. The states qualified as `<<error>>` states can be used in the model in order to express the satisfaction or the non satisfaction of a safety property. Observers may access any part of UML model's state (object attributes and states, signal queues) and they may use clocks to express timing properties. So, special events have been defined for observers in order to meet their purpose, events related to:

- signal exchange: **send**, **acceptsignal**, **receivesignal**;
- operation calls: **invoke**, **receive** (reception of call), **accept** (start of the actual processing of call), **invokereturn** (sending of a return value), **receivereturn** (reception of the return value), **acceptreturn** (consumption of the return value);
- execution of actions or transitions: **start**, **end**, **startend**;
- timers: **occur**, **timeout**, **set**, **reset**.

The trigger of an observer transition is a **match** clause specifying the type of the event (previously presented), some related information (for example the operation name) and observer variables that may receive related information (variables receiving the values of the signal/operation call parameters).

For further details on the time extension and observers the reader is referred to [18].

3 Overview of the Composite Structures

Composite structures have been introduced in the UML standard starting with version 2.0. They refer to "a composition of interconnected elements, representing run-time instances collaborating via communication links to some common objectives" ([12] pp. 161). Composite structures are a big evolution in modelling a system and are often used for the hierarchical representation of real-time embedded systems.

A *composite structure* is formed by inner components, that are called *parts*, and communication paths, that are called *connectors* or *links*. *Parts* are instances of classes with predefined role and they usually are in a fix number within the composite structure. In Figure 1 parts are represented by the instances of Keypad, Display, CashUnit, CardUnit,

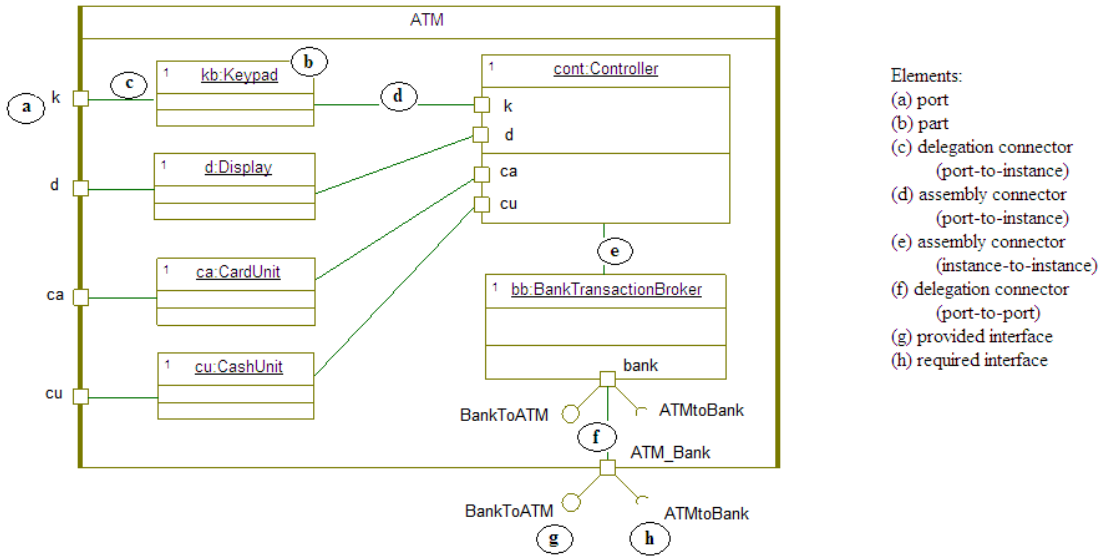

Figure 1: Composite structure example

Controller and BankTransactionBroker. *Links* connect inner components (e.g. **e** in Figure 1), an inner component with a port (e.g. **c, d**) or two ports (e.g. **f**), so that these elements can communicate between them via signals or operation calls. Such links can transport signals between elements that know how to answer to them. A link can be the realization of an *association* (especially in the case of a connector between two parts), but this is not mandatory. The UML standard classifies links in two categories: *delegation links* which connect the composite structure with one of its components (part or port of a part) and *assembly links* which connect two components between them. *Delegation links* can be separated in *outbound delegation links* and *inbound delegation links*, depending of connector's direction (if it is oriented to the outside environment or, correspondingly, to the inner structure).

A *port* (e.g. the elements **k, d, ca, cu** from ATM and Controller, **bank** from BankTransactionBroker and **ATM_Bank** from ATM in Figure 1) is an interaction point between its owner and the outside environment. Every port has a contract, given by classifiers (interfaces or classes), which allows it to handle known requests by forwarding them in the needed direction. These requests can be incoming requests from the environment (the port *provides* the interface; e.g. **g** from Figure 1) or outgoing requests to the environment (the port *requires* the interface; e.g. **h** from the same Figure).[3]

Anyhow composite structures, as presented in UML standard, are ambiguous. For example, every connector links two entities which can be either ports or components (parts). In both cases, the two entities are typed[4]. In addition, the modeller can specify that the

[3] *Provided* and *required* are defined from the component owning the port point of view.

[4] For a part the type is given by the class whose instance it is and for a port the type is given by its

connector *realizes* an *association*. It is clear that, in general, connecting entities of arbitrary types does not make sense, and there should be clear compatibility rules (based on types, link direction, etc.) specifying what are the well formed structures. However, these type compatibility rules for connectors are not detailed in UML. The standard merely states that "the connectable elements attached to the ends of a connector must be compatible." and that "what makes connectable elements compatible is a semantic variation point" ([12] pp. 175-176). Various causes of ambiguity, such as the existence of several connectors starting from a same end-point, are not even mentioned.

Our purpose is to define a rule set in order to disambiguate the composite structures so that we shall have a clear a coherent executable semantics. Therefore, we had extended the OMEGA UML Profile to cover unambiguous composite structures by setting well-formedness constraints and by clarifying the run-time behaviour of these structures. A formalization of the rules provided is needed for proving the type-safety of our system.

4 The extended OMEGA Profile: well-formedness and typing rules

The extended OMEGA UML Profile (called OMEGA2) introduces an expressive and unambiguous set of constructs for modelling hierarchical structures, with an operational semantics that integrates in the existing execution model of OMEGA. The typing system and the consistency rules we have formulated can be applied to other component-based models like SysML ([10]) or MARTE ([11]).

4.1 Bidirectional ports

A first typing problem comes from the fact that in UML the ports are bidirectional, i.e. they can specify a set of allowed incoming requests (the *provided* interfaces) and a set of allowed outgoing requests (the *required* interfaces). This is represented in the model as follows: all the interfaces that are directly or indirectly *realized* by the type of the port (its contract) are considered to be *provided* interfaces. The *required* interfaces are those interfaces for which there exists a *Dependency* stereotyped with `<<Usage>>` between the port type (or one of its supertypes) and the respective interface(s). Figure 2-a shows a simple example of bidirectional port.

The type of the port `port_0` in Figure 2-a is I. However, the fact that the port is bidirectional raises typing problems, which are apparent in the following situations:

- When `port_0` is used by A to send out requests conforming to interface J, by an action such as "`port_0.op2()`". In this case, `port_0` has to be treated by the type system as an entity of type J, although it is declared of type I.
- When one wants to specify behaviour of `port_0` by a state machine[5]. Then the state

contract.

[5]This is deemed possible by the UML 2.x standard [12], but without further detail.

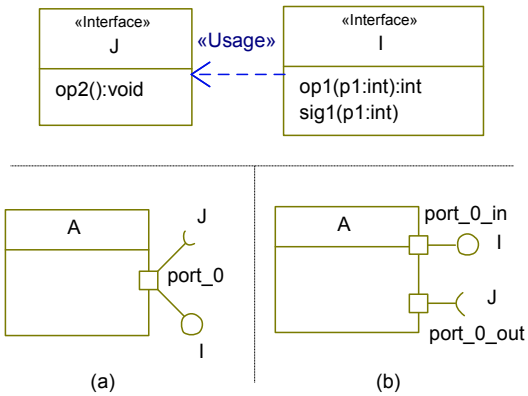

Figure 2: (a) - Example of a biriderctional port, (b) - Equivalent in OMEGA2

machine has to handle requests coming from both directions, i.e. requests conforming both to I and to J.

These typing inconsistencies are not addressed by the UML standard. When we translate UML models into their IF description (or other implementation languages) they raise homologous problems for the typing of the actual object that will represent the port. A general solution, based on qualifying the types (I, J) with the corresponding directions (*in*, *out*) and on allowing the port entity to comply to multiple types, is possible but it greatly complicates the type checking of UML models.

For these reasons, the solution we adopt in OMEGA2 is to forbid bidirectional ports. This is possible because any bidirectional port can be split in two unidirectional ports, like in the example from Figure 2-b, although it can be argued that it leads to less convenient models.

Syntactically, an unidirectional outgoing port specifying a *required* interface J (such as `port_0_out` from Figure 2-b) will be represented as a port typed with J and stereotyped with `<<reversed>>` (to distinguished it from a port *providing* J).[6]

4.2 Directionality rules

A second typing problem is raised by connectors. No compatibility rules for links are given by the standard. Before presenting type compatibility issues for links, some simple directionality rules must be observed by well-formed structures:

Rule 1 *If a* delegation *link exists between two ports, the direction* (provided *or* required) *of the ports must be the same.*

[6]Note that a mechanism identical to the «`reversed`» stereotype is supported by the IBM Rhapsody tool [13], including support for graphical representation using the standard *required interface* symbol of UML like in Figure 2-b. For editing convenience, the Rhapsody representation is also supported by the IFx2 tools.

Rule 2 *If an* assembly *link exists between two ports, one of the ports (the source) must be a «reversed» port (required) and the other (the destination) must be a normal port (provided).*

Rule 3 *If a link is typed with an association, the direction of the association must be conform to the direction of the link (derived from the direction of the ports at the ends).*

Rule 1 restricts the links that can be used and we summarize here which connectors are accepted in OMEGA2:

1. Part - Part link ⇒ assembly link, needs to be typed with an association[7]
2. Port - Port link
   - 2.1 One port owned by the composite structure, the other one owned by a part
     port required - port required ⇒ outbound delegation link
     port provided - port required ⇒ forbidden
     port required - port provided ⇒ forbidden
     port provided - port provided ⇒ inbound delegation link
   - 2.2 Both ports are owned by parts
     port required - port required ⇒ forbidden
     port provided - port required ⇒ assembly link
     port required - port provided ⇒ assembly link
     port provided - port provided ⇒ forbidden
3. Part - Port link
   - 3.1 Port owned by a part
     part - port provided ⇒ assembly link, needs to be typed with an association
     part - port required ⇒ assembly link
   - 3.2 Port owned by the composite structure
     part - port provided ⇒ inbound delegation link
     part - port required ⇒ outbound delegation link, needs to be typed with an association

The third rule introduces more constraints in the profile by establishing a correspondence between the direction of a connector typed with an association and the related association. These types of connectors need to be treated carefully so that by typing a link with an association, the direction in which it transports the messages does not become inconsistent and therefore the composite structure is not well-formed. This rule can be expanded in three cases:

[7]The need of typing a link with an association is given by the fact that a component has to know how to address the connector (see Rule 5 later on).

- The association is navigable at both ends. This type of association is accepted only for a link that connects two parts and the types of each end of the link and association must be compatible.
- The association is navigable only at one end. Then the types at each end of the link and the association should be compatible. This restricts us the associations that we may have:
  - For a link between two parts, the accepted associations are associations between two classes, a class and an interface or two interfaces.
  - For a link from a part to a port, the accepted associations are between a class and an interface pointing to the interface or between two interfaces.
  - For a link from a port to a part in this direction, only the association between two interfaces is accepted.
  - For a link between two ports, only the association between two interfaces is accepted.
- The association is not navigable at both ends then the connector is not well-formed.

The end of the link is compatible with the corresponding end of the association means:

- If the end of the link is a part and the association end is a class then the association end has to be equal or a supertype for the link's part type.
- If the end of the link is a part and the association end is an interface then link's end type has to realize directly or indirectly the association's end type.
- If the end of the link is a port and the association end is an interface then the port has to provide/require the association end.

We have mentioned as notion the *direction of a link*. We can establish the direction based on a link's type and we can define the following. A connector starts from a port providing interfaces if:

- It is an inbound delegation link between provided ports and the port is owned by the composite structure;
- It is an inbound delegation link between a provided port and a part.

A connector starts from a port requiring interfaces if:

- It is an outbound delegation link between required ports and the port is owned by the inner component;
- It is an assembly link between provided-required ports;
- It is an assembly link between a part and a required port.

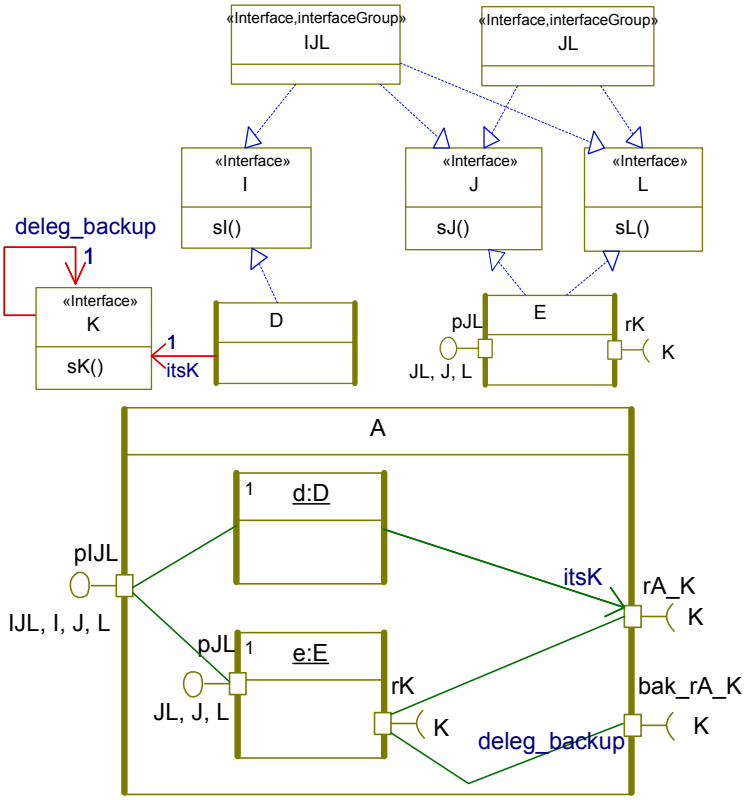

Figure 3: Connection rules in composite structures

A connector starts from an inner component if:

- It is an assembly link between a part and a provided port;
- It is an outbound delegation link between a part and a required port;
- It is an assembly link between parts.

Taking as a running example the Figure 3 we can express the reason behind these rules. The example is a composite A with two sub-components of types D and E, one using ports for communication (E) and one not (D). For both sub-components there are incoming links (links from port $pIJL$ of A) and outgoing links (links to ports rK and bak_rA_K of A).

So, Rule 1 forbids putting a connector, for example between $pIJL$ and rK, since the direction of the connector would be ambiguous. Rule 3 forces the direction of a connector to be coherent with the direction of the realized association, like in the case of the link between d and rA_K (realizing association $itsK$).

These three rules allow us to have an overview on the composite structure and its behaviour. We can follow the flow of requests based on link's direction and establish which component reacts such that the main goal is achieved.

4.3 Type coherence rules

Before presenting the type system for connectors and type-based rules, we need to introduce some notions: *interface groups*, *default delegation associations* and *set of transported interfaces*.

Interface groups. Let us note that it is sometimes necessary to declare several provided or required interfaces for one port (for example, $pIJL$ of A which provides interfaces I, J and L, see Figure 3). In UML, this is done by declaring a new interface that inherits from these interfaces and by using this new interface as the port type (IJL in Figure 3). However, such interfaces are artificial syntactic additions to the model, and they should not be taken into consideration by the link compatibility rules stated in the following. In our example, d and e only realize interfaces I and respectively J and L, so interface IJL is irrelevant for the semantics of the model. In OMEGA2, such interfaces must be stereotyped with «`interfaceGroup`» to distinguish them from meaningful ones, as shown in the upper part of Figure 3.

Default delegation associations. The default behaviour of a port is to forward requests from one side to the other according to its direction: to the environment, if it is a *required* port, and to its owner, if it is a *provided* port. The minimum information needed by the port is, for each provided/required interface, which the destination should be. For example in Figure 3, port $pIJL$ needs to know (and be able to refer to) the destination of requests belonging to interface I (here, d) and the destination of requests belonging to J or L (here, pJL of e). Similarly, rK needs to know that the destination of outgoing requests is by default rA_K.

It follows that, for each provided/required interface, the port has to possess an association designating to which the port should forward requests belonging to that interface. In OMEGA2, every interface type I has by default an association called $deleg\_I$ pointing to itself, used for this purpose (for modelling convenience, the semantics considers they exist by default if they are omitted in the model). These associations are used to define the forwarding semantics of ports, described later on.

The dynamic type of a connector. The type of a connector determines what type of invocations (signals or operation calls) can travel through the connector and how port behaviour descriptions refer to the connector. In general, in the case of a connector *originating*[8] from a port (i.e., not directly from a part), its type can be derived from the type of the entities situated at its two ends and does not necessarily need to be statically specified using an *association*. The following notion defines the dynamic type of the connector:

Definition 1 [Set of transported interfaces]. For a connector starting from a port, the *set of transported interfaces* is defined as the *intersection* between the two sets of interfaces provided/required at the two ends of the link.

[8]According to link directionality, as explained in Section 4.2.

As the ends of a link can be either ports or components, the meaning of provided/required interfaces is defined for each case:

- For a *Port*, the set of required/provided interfaces is the set containing the *Port*'s type and all its supertypes, without all the interfaces stereotyped as «`interfaceGroup`».
- For a component, the set of provided interfaces is the set of all interfaces directly or indirectly realized by the component's class.

According to this definition, the set of transported interfaces for the links in Figure 3 are as follows[9]:

- For link $pIJL$ to d the set is $\{I\}$.
- For link $pIJL$ to pJL the set is $\{J, L\}$.
- For link rK to rA_K the set is $\{K\}$.

Let us note that the link from $pIJL$ to d given as example above could have been statically typed with association $deleg\_I$, because the set of transported interfaces $\{I\}$ only contains one element. However, in the general case when the derived set contains several interfaces (like for example the link between $pIJL$ and pJL which transports $\{J, L\}$), statically typing a link with an association is not necessary and may be restrictive.

If a static type is specified, it must be compatible with the dynamic type, as stated in the following rule:

Rule 4 *If a link outgoing from a port is statically typed with an association, then the association is necessarily directed (cf. Rule 3) and the type pointed at by the association must belong to the* set of transported interfaces *for that link.*

On the example in Figure 3, Rule 4 implies that, for example, if the link $pIJL$ to pJL is statically typed with an association then the association must point at either J or L. But this restricts the set of requests forwarded through the link to only those requests which belong to the pointed interface (J or L), therefore the behaviour is restricted compared to a dynamically typed link.

While the type for a connector starting *from a port* does not need to be statically specified as it can be derived as shown before, if the connector starts directly *from a component* (and not from a port) then the static type *must* be specified:

Rule 5 *If a link originates in a component, then the link* must *be statically typed with an association, and the type of the entity at the other end of the link must be compatible with (i.e. be equal or a subtype of) the type at the other end of the association.*

[9]Link d to rA_K starts from the part d and therefore the set of transported interfaces cannot be computed; moreover the link has to be statically typed (see Rule 5 later on).

In Figure 3, only the link from d to rA_K is in this case; the link has indeed to be typed (here, with $itsK$) or otherwise the component would have no means to refer to it for communication.

Finally, a link is meaningful only if it can transport some requests:

Rule 6 *The set of transported interfaces for each link should not be void.*

The above rules allow us to specify exactly what requests (signals and operation calls) can travel through connectors by defining compatible interfaces for each component.

4.4 Port behaviour rules

In OMEGA2, the default behaviour of a port is to forward requests from one side to the other, depending on the port's direction. Each request (signal or operation call) will be forwarded to a destination which depends on the interface to which the signal or operation belongs, using the default *deleg* associations above described. For example, the default forwarding behaviour of port $pIJL$ from Figure 3 can be described by the state machine in Figure 4-a[10].

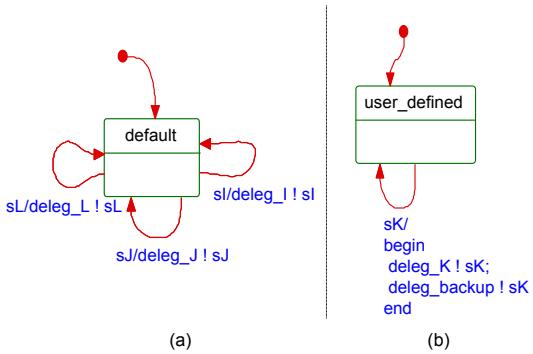

Figure 4: (a) - Default state machine for port $pIJL$, (b) - User-defined machine for port rK

The default behaviour is unambiguous only if for any interface, the entity to which the corresponding *deleg* association points at is clear. Therefore, the following rules are necessary:

Rule 7 *If several* non-typed *connectors start from one port, then the sets of interfaces transported by each of the connectors have to be pairwise disjoint.*

[10] `deleg_I!sI` is the OMEGA2 syntax for the action of sending signal `sI` to the destination `deleg_I` (if the signal has formal parameters and no actual parameters are specified in the sending action, the actual values that will be sent are those ones received at the last *reception* – here the one that triggered the transition).

The last rule does not forbid the case where a port is connected to n entities that provide or require the same interface I ($n > 1$): it states that in this case at least $n-1$ connectors have to be explicitly typed with associations. The one connector which is not explicitly typed, if it exists, is implicitly typed with *deleg_I*. In the example from Figure 3, port rK of e is in this situation: it has two links to two ports (rA_K and *bak_rA_K*), both typed with the same interface (K). According to Rule 7, one of the links has to be explicitly typed; here, the second one is statically typed with the association *deleg_backup*.

The default port behaviour may be redefined by attaching a state machine to the port's type. In OMEGA2, this state machine may use the implicitly typed connectors (accessed via the default *deleg* associations), as well as the explicitly typed connectors (via their defining association). In Figure 4-b we show an example of port behaviour for port rK (from Figure 3), which duplicates every sK signal on both the default connector (*deleg_K*, communicating with rA_K) and the secondary connector (*deleg_backup*, communicating with *bak_rA_K*).

In addition, for completeness of the port behaviour, we require the following:

Rule 8 *The union of the sets of interfaces transported by each of the connectors* originating *from a port P must be equal to the set of interfaces provided/required by P.*

Applied for example to port $pIJL$ from Figure 3, this rule says that the two links originating from the port must transport, together, the entire set of interfaces provided by the port, i.e. $\{I, J, L\}$ (remember that IJL is an «`interfaceGroup`» and does not count in type checks).

4.5 Concurrency model and observers

The concurrency model is left open in UML. The previous version of OMEGA defined a particular concurrency model, based on the standard UML notion of *active* and *passive* classes. Due to the choice of partitioning the object space in activity groups attached to active objects, certain forms of simple resource sharing and synchronisation generated quite complex models, as sharing could only be achieved via an explicitly modelled resource manager – an active object. In order to overcome this problem, a new kind of passive class can be defined in OMEGA2 (using the stereotype «`protected`»).

Protected objects are passive objects that do not belong to one activity group but rather are shared between the groups. They work in the same way as Ada protected objects [1]. Like in Ada, protected objects are a synchronization mechanism. They provide *functions* (which may only read but not modify object attributes) that can be executed concurrently, and *entries* that are executed in mutual exclusion from each other and from functions (this corresponds to the classical readers-writers pattern). In addition, an entry has a guard; a call to an entry from a thread (activity group) will wait before beginning the execution until the guard is true. Our model of protected objects is slightly simplified (more non-deterministic) compared to the *eggshell* model of Ada [1] and therefore suppresses the need for *procedures* existing in Ada: a procedure can be seen as an entry with guard `true`.

The OMEGA2 concurrency model therefore distinguishes three kinds of classes: active, passive and protected. Since every passive object is considered to *belong* to an active object, in the sense that its behaviour is executed on the execution thread of its owner, some rules are necessary to avoid confusing configurations in composite structures.

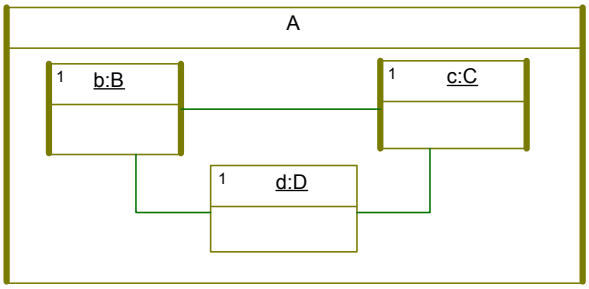

Figure 5: Forbidden composite structure.

For example, the composite structure in Figure 5 shows, on the same level, two active objects (b and c), which have their own *activity group*, and a passive object (d) which belongs to the group of its creator (instance of A). This kind of structure is forbidden. If the desired semantics is to have a shared passive object d, then d may be declared as «`protected`» and the structure becomes valid.

Rule 9 *A passive class may define a composite structure formed only of passive classes.*

Rule 10 *An active class may define a composite structure formed of either only passive classes, or of a combination of active and protected classes.*

As an extension to the original profile, an observer can also define a simple composite structure. Composite observers have proved to be a way for making more compact the specification of some complex verification properties. The well-formedness rule is:

Rule 11 *If an Observer defines a composite structure, the components must also be instances of Observers.*

5 Translation to IF

The mapping between the OMEGA Profile and the IF language is based on the principles explained in this section.

Every UML class is mapped to a process with a local variable for each attribute or association of the class. Inheritance between classes is translated by the duplication of each inherited attribute in the processes corresponding to subclasses. Operations are defined by signals, while statemachines are signals are translated almost syntactically to IF. For the time extension *clocks* exist as a predefined type in IF and *timers* are translated using a

clock and a timer process sending *timeout* signals. For further details the reader is referred to [19].

The translation of composite structures is based on the principle that the modelling elements involved in composite structures, namely ports and connectors, should be handled as first class language citizens. This means that we refrain from flattening the model during compilation and hard-wiring all the communication paths (something that is done, for example, in certain SDL compilers). Concretely, each port instance is implemented as an IF process instance (whose behaviour corresponds to the routing behaviour described in Section 4.4) and each connector is represented by attributes in the end-points (in ports or in components), corresponding to the association defining the connector (the default *deleg* association or the explicitly specified one).

In this setting, a UML composite structure diagram is used simply as an initialization scheme for instantiating components and ports and for creating links. A composite structure is therefore translated to a constructor.

As a consequence of the translation sketched above, a signal or operation call sent through a connector chain will pass through several objects (the intermediate ports) before reaching the destination. In order to avoid the state space explosion problem due to the interleaving of such "forwarding" actions, the translator defines a *total priority order* between these actions. Thus, even if several signals are in transit on connector chains, only one forwarding action (belonging to the enabled port with the highest priority) will be enabled at any given time. This yields an increase of state space due to connector chains which is linear in the length of the chain, instead of combinatorial explosion. Note that starvation of lower priority actions is not possible since, in any state, eventually all signals that are in transit through connectors will arrive at destination and the rest of the system will be able to make progress. Moreover, this abstraction is made without any loss of generality, since all the possible interleavings at the level of component transitions (which is the observable level) remain feasible. The implementation of the abstraction is made very easy by the dynamic priority mechanism of IF .

Another element that is added in the second version of the OMEGA profile is *protected* classes. Compared to normal passive classes, protected classes add the classical *readers-writers* synchronization protocol for functions and entries. The readers-writers protocol implemented in our translation is a variant of the classical solution that may be found in many textbooks (e.g., [4]). The implementation is however facilitated by the fact that the IF language offers mutually exclusive and atomic *transitions* by default, and transitions can specify conditional waiting simply using guard conditions.

For structured observers, the same mapping as for composite structures is applied.

6 OCL Formalization

The rules disambiguating composite structures also implemented in the OMEGA2 compiler are formalized in OCL to verify that UML models comply with our profile. The OCL code was developed in Topcased OCL Environment [22].

For our formalisation we have defined helper functions for accessing:

- type of elements connected with a link: *has2Ports*, *has2Parts*, *has1PartAnd1Port*, *has1PartWithPort*, *has2PartsWithPort*;
- the connected elements: *part1*, *part2*, *port1*, *port2*;
- association's properties: *isTyped*, *isBidirectional*, *isNotNavigable*, *isEnd1Navigable*, *associationEnds*, *isClassClassAssociation*, *isInterfaceInterfaceAssociation*, *associationStartPoint*, *associationStartPointType*, *associationEndPoint*, *associationEndPointType*, *isClassInterfaceAssociation*;
- port's type: *isReversed*;
- types of classifiers: *isInterfaceGroup*, *isInterface*, *isProtected*, *isObserver*.

The definition of these functions, together with the invariants presented in this Section, can be found in the Appendix.

For the formalization of Rule 1 and Rule 2 we define a function that will compute the exact link type based on the classification presented in Section 4.2. The OCL invariant corresponding to these two rules becomes the verification for each connector in the model if its type is not *forbidden*:

```
context Connector

-- Definition of link's type
def: linkType : String =
  if has2Parts
    then 'assembly link between parts'
  else
    if has2Ports
      then if has1PartWithPort
             then if not port1.isReversed and not port2.isReversed
                    then 'inbound delegation link between provided ports'
                  else if port1.isReversed and port2.isReversed
                         then 'outbound delegation between required ports
                             '
                       else 'forbidden'
                       endif
                  endif
           else if (port1.isReversed and port2.isReversed) or
                   (not port1.isReversed and not port2.isReversed)
                  then 'forbidden'
                else  'assembly between provided-required ports'
                endif
           endif
    else
      if has1PartWithPort
        then if not port1.isReversed
```

```
                then 'assembly link between part and provided port'
              else 'assembly link between part and required port'
              endif
        else if not port1.isReversed
              then 'inbound delegation link between part and provided port
                 '
            else 'outbound delegation link between part and required port'
            endif
        endif
      endif
    endif

  -- Rule 1 and Rule 2
  inv LinkType: self.linkType <> 'forbidden'
```

For the formalization of the third rule we need to verify the compatibility between the association end (of the association typing the link) and the corresponding link end (i.e. the compatibility has to be verified between the start point for both link and association and for the end point). As explained in Section 4.2, this resumes to verify the inclusion of the association's end type in link's end type (realized interfaces or superclasses). We define functions that verify if a link starts from a port or a part as already presented (*isStartingFromProvidedPort*, *isStartingFromRequiredPort*, *isStartingFromPort*, *isStartingFromPart*), and for each link which is its starting point and its ending point (*linkStartPort*, *linkEndPort*, *linkStartPart*, *linkEndPart*). Since the formalization of the following rules consists in computing the provided/required interfaces for a port and a component and since the same calculus can be used for Rule 3, we shall continue by computing the needed sets.

We continue with the calculus of the provided/required interfaces for a port and the interfaces provided by a component. In the case of a port, the set of realized interfaces is given by the set of provided interfaces without those stereotyped with <<interfaceGroup>>.[11] Please note that interfaces stereotyped <<interfaceGroup>> are artificially added to the model and they should not be taken into consideration in our formalization.

```
context Port

-- Definition of interfaces realized by a port
def: interfaces : Set(Classifier) = self.provided->reject(
    isInterfaceGroup)
```

Before computing the set of realized interfaces by a class (the type of a part), we have to compute recursively the list of parents. We suppose that our model is well-formed and has no cycles.

[11] Required interfaces are modelled with reversed ports and are therefore also accessed using `provided`.

```
context Classifier

-- Definition of classifier's parents recursive computation
def: getParentsRec : Set(Classifier) =  self.general->union(self.general
    ->iterate(p:Classifier; res:Set(Classifier)=Set{}| res->union(p.
    getParentsRec)))
```

For a class, the set of provided interfaces is the set of realized interfaces summed with the set of parents for each realized interface and summed with the set of provided interfaces for each parent of our class, without those stereotyped with <<interfaceGroup>>.

```
context Class

-- Definition of all interfaces directly realized by a class
def: iRealizations : Set(Classifier) = self.interfaceRealization.contract
    ->asSet()

-- Definition of interfaces provided by a class directly or indirectly
    realized (used in the case of a link not typed by an association)
def: interfaces : Set(Classifier) =
  iRealizations->union(iRealizations->iterate(i:Interface; res:Set(
     Classifier)=Set{}| res->union(i.getParentsRec)))
  ->union(self.getParentsRec->iterate(c:Class; res:Set(Classifier)=Set{}|
      res->union(c.interfaces)))
  ->reject(isInterfaceGroup)
```

In the case of a link typed with an association which has as an end an interface, the set of provided interfaces is the set of the interface to which it points summed with its parents and without those interfaces stereotyped with <<interfaceGroup>>.

```
context Interface

-- Definition of interfaces provided by an interface (used in the case of
     a link typed by an association pointing to an interface)
def: interfaces : Set(Classifier) = self.oclAsType(uml::Classifier)->
    asSet()->union(self.getParentsRec)->select(not isInterfaceGroup and
    isInterface)
```

Because of the mishandling of polymorphic functions in OCL, we need to define explicitly the polymorphism of the function *interfaces* on the subtypes of *Type* (*Class* and *Interface*).

```
context Type

-- Determines the set of provided interfaces by a class or an interface
def: interfaces : Set(Classifier) =
```

```
  if self.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces
  else
    self.oclAsType(uml::Class).interfaces
  endif
```

We compute the set of transported interfaces as the intersection of provided interfaces of both ends and we formalize Rule 6: the cardinal of the set of transported interfaces should be at least equal to one. We need to remark that this set is computed for links starting from a port (typed or not typed with an association) and it is not computed in the case of a part-part connector.

```
context Connector

-- Definition of the set of transported interfaces
def: setTransportedInterfaces : Set(Classifier) =
  if has2Parts
    then Set{OclInvalid}
  else if has2Ports
         then if isTyped
                then if isStartingFromPort(port1)
                       then (port1.interfaces) -> intersection(
                          associationEndPointType.interfaces)
                     else (port2.interfaces) -> intersection(
                        associationEndPointType.interfaces)
                     endif
              else (port1.interfaces) -> intersection(port2.interfaces)
              endif
       else
         if isTyped
           then (port1.interfaces) -> intersection(
              associationEndPointType.interfaces)
         else (port1.interfaces) -> intersection(part1.type.interfaces)
         endif
       endif
  endif

-- Rule 6
inv SetOfTransportedInterfacesNonEmpty: self.setTransportedInterfaces->
    size()<>0
```

In order to formalize Rule 4 and Rule 5, we define the compatibility between two classes and between a class and an interface as the inclusion of association's end type in the set of provided interfaces or in the set of parents. It is followed by the compatibility between a port and an interface as the inclusion of all realized interfaces by the association's end type in the set of provided/required interfaces of the port.

```
context Classifier

-- Verifies if the current classifier is compatible with the one given as
    parameter
def: isCompatible(c:Classifier) : Boolean =
  if self.oclIsKindOf(uml::Interface) and c.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces->includes(c)
  else
    if self.oclIsKindOf(uml::Class) and c.oclIsKindOf(uml::Interface)
      then self.oclAsType(uml::Class).interfaces->includes(c)
    else
     (c->asSet()->union(c.getParentsRec))->includes(self)
    endif
  endif

context Type

-- Verifies if the link end's type (the type of a part) is compatible
   with the association end's type given as parameter
def: isCompatible(t:Type) : Boolean = self.oclAsType(uml::Classifier).
   isCompatible(t.oclAsType(uml::Classifier))

context Port

-- Verifies if the current port is compatible with the association end's
   type given as parameter
def: isCompatible(t:Type) : Boolean = self.interfaces->includesAll(t.
   oclAsType(uml::Interface).interfaces)
```

Rule 4 states that for a link starting from a port and typed with an association, the association must be directed (unidirectional) and the interface pointed by the association has to be included in the set of transported interfaces. We include here Rule 3, which adds that the direction of the link has to be conforming to the direction of the association, as defined in Section 4.2. We define a function (*linkStartingFromPortVerification*) that verifies if for a link typed with an interface-interface association (the only association accepted for a connector starting from a port) the corresponding start points and end points are compatible and that the interface to which it points is included in the set of transported interfaces.

```
context Connector

-- Verifies if a link starting from a port and typed with an association
   has the same direction with the association and the interface pointed
   is included in the set of transported interfaces
def: linkStartingFromPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isInterfaceInterfaceAssociation
```

```
        then if has1PartAnd1Port
              then linkStartPort.isCompatible(associationStartPointType)
                  and linkEndPart.type.isCompatible(
                  associationEndPointType) and
                    setTransportedInterfaces->includes(
                        associationEndPointType.oclAsType(uml::Classifier))
             else linkStartPort.isCompatible(associationStartPointType)
                 and linkEndPort.isCompatible(associationEndPointType) and
                  setTransportedInterfaces->includes(
                     associationEndPointType.oclAsType(uml::Classifier))
             endif
    else false
    endif
  endif
```

Then the OCL invariant corresponding to these two rules verifies that for each link in the model starting from a port and typed with an association the compatibility stated above is verified.

```
context Connector

-- Verifies if a link starting from a port is well-formed
def: linkStartingFromPort : Boolean =
  if (not isStartingFromPart) and isTyped
    then linkStartingFromPortVerification
  else true
  endif

-- Rule 3 and Rule 4
inv LinkStartingFromPort: self.linkStartingFromPort
```

Rule 5 completes Rule 3, by adding that all connectors starting from a part have to be typed with an association and if the association is bidirectional (the only bidirectional association accepted is the association between two classes that may type only the link that connects two parts) it has to be compatible with the link in a direction. For a unidirectional association that types the link, we need to have the compatibility between the corresponding ends (link's start part with association's start point and link's end part/port with association's end point). This is expressed by the below functions (*linkPartPartVerification*, *linkPartPortVerification*), which make the difference between a link that connects two parts (which accepts all kinds of associations) and the link that connects a part with a port (which accepts only the association between two interfaces or between a class and an interface).

```
context Connector

-- For a link between two parts verifies if the ends are compatible with
   the corresponding ends of the accepted association
```

```
def: linkPartPartVerification : Boolean =
  if isNotNavigable
    then false
  else
    if isBidirectional
      then (linkStartPart.type.isCompatible(associationStartPointType)
          and linkEndPart.type.isCompatible(associationEndPointType)) or
               (linkStartPart.type.isCompatible(associationEndPointType)
                  and linkEndPart.type.isCompatible(
                  associationStartPointType))
    else (linkStartPart.type.isCompatible(associationStartPointType) and
       linkEndPart.type.isCompatible(associationEndPointType))
    endif
  endif

-- For a link between a part and a port verifies if the ends are
   compatible with the corresponding ends of accepted association
def: linkPartPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isClassClassAssociation
      then false
    else
      if isInterfaceInterfaceAssociation
        then (linkStartPart.type.isCompatible(associationStartPointType)
           and linkEndPort.isCompatible(associationEndPointType))
      else
        if isClassInterfaceAssociation
          then linkStartPart.type.isCompatible(associationStartPointType)
              and linkEndPort.isCompatible(associationEndPointType)
        else false
        endif
      endif
    endif
  endif
```

The invariant for Rule 3 and Rule 5 verifies that each connector in the model starting from a part is typed with an association and the direction of the association is compatible with the direction of the link:

```
context Connector

-- Verifies if a link starting from part is typed with an association and
    if it is well-formed
def: linkStartingFromPart : Boolean =
  if isStartingFromPart
    then if has2Parts
           then isTyped and linkPartPartVerification
         else isTyped and linkPartPortVerification
```

```
        endif
  else true
  endif

-- Rule 3 and Rule 5
inv LinkStartingFromPart: self.linkStartingFromPart
```

We formalize now the rules for port behaviour. The default behaviour is that the port forwards the requests received according to its direction: to the environment if it is a *required port* and to the component that owns it if it is a *provided port*. This means that the port knows how to respond to any received request and also which is the destination of the request.

The context in our formalization becomes the *Port* and we define functions that give all the connectors (typed or not with an association) starting from the port (*connectors*, *connectorsNotTyped*, *connectorsStartingFromPort*) and that verify if the port has connectors (typed or not with an association) starting from it (*hasConnectors*, *hasTypedConnectors*, *isStartingPort*).

The relation behind the first rule concerning port's behaviour states that the sets $A_1, A_2, ..., A_n$ are *pairwise disjoint* if and only if $\text{card}(A_1 \cup A_2 \cup ... \cup A_n) = \text{card}(A_1) + \text{card}(A_2) + ... + \text{card}(A_n)$. The function *unionSetForTransportedInterfacesOnLinks* computes the left hand side of the equality, and the function *noOfTransportedInterfacesOnLinks* computes the right hand side of the expression.

```
context Port

-- Determines the union of the sets of transported interfaces on each
   link starting from the port
def: unionSetForTransportedInterfacesOnLinks(withType:Boolean) : Set(
   Classifier) =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; s:Set(
     Classifier)=Set{} | s->union(c.setTransportedInterfaces))

-- Determines the sum of the number of transported interfaces on each
   link starting from the port
def: noOfTransportedInterfacesOnLinks(withType:Boolean) : Integer =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; i:
     Integer=0 | i + (c.setTransportedInterfaces->size()))
```

Then Rule 7 becomes the verification of the equality stated above:

```
context Port

-- Definition of pairwise disjoint sets of transported interfaces
context Port
def: isPairwiseDisjoint : Boolean =
  if self.connectorsStartingFromPort(false)->size() >= 2
```

```
    then  self.unionSetForTransportedInterfacesOnLinks(false)->size() =
       self.noOfTransportedInterfacesOnLinks(false)
  else true
  endif

-- Rule 7
inv PairwiseDisjoint: self.isPairwiseDisjoint
```

For Rule 8 we will use the union of the sets of transported interfaces computed above and we will test its equality with the set of provided/required interfaces by the port.

```
context Port

-- Verifies if the union of sets of transported interfaces is equal to
   the interfaces provided/required
def: isComplete : Boolean =
  if isStartingPort
    then unionSetForTransportedInterfacesOnLinks(true) = self.interfaces
  else true
  endif

-- Rule 8
inv Completeness: self.isComplete
```

For the two rules concerning the execution model for composite structures we will reason on the number of active, passive and protected components given by the functions *noOfComponents*, *noOfActiveComponents*, *noOfPassiveComponents*, *noOfProtectedComponents* and *isComposite*.

Rule 9 says that if a composite structure is passive then it is well-formed if and only if the number of passive parts this owns is equal with the total number of parts. Rule 10 says that if a composite structure is active then it is well-formed if and only if or the number of passive parts is equal to the total number of parts or the sum between the number of active parts and the number of protected parts is equal to the total number of parts.

```
context Class

-- Definition of a well-formed class
def: isWellFormed : Boolean =
  if self.isActive and isComposite
    then  if noOfActiveComponents + noOfProtectedComponents =
       noOfComponents or
            noOfPassiveComponents = noOfComponents
          then true
       else false
       endif
  else
    if (not self.isActive) and (not isProtected) and isComposite
```

```
      then if noOfPassiveComponents = noOfComponents
             then true
           else false
           endif
    else OclInvalid
    endif
  endif

-- Rule 9 and Rule 10
inv CompositeStructure: self.isWellFormed <> false
```

The last rule regarding the simple composite observers is also formalized with the means of the number of observer parts. This rule is equivalent to: the number of parts of a composite observer is equal to the number of observer parts (*noOfObservers*) of the same composite structure.

```
context Class

-- Definition of a well formed observer
def: isObserverWellFormed : Boolean =
  if self.isObserver and isComposite
    then noOfComponents = noOfObservers
  else true
  endif

-- Rule 11
inv CompositeObserver: self.isObserverWellFormed <> false
```

7 Evaluation

To validate the approach, we evaluated the rules on several complex models. The most complex example we used is a model of the solar wings deployment system of the ATV[12] provided by Astrium Space Transportation. The model features a 3-level hierarchical architecture with 37 classes (7 composite ones), 93 active objects at runtime and approximately 380 ports and 200 connectors.

The OCL formalization was applied on the model in order to test model compliance with the OMEGA2 profile and to search for modelling errors. Since the original model had not been built for simulation or verification, the first issue pointed out by the rules was ports and connectors were untyped. The corrective action consisted in defining a total of 26 interfaces, and using them for specifying port contracts. Only a few ports in the original model were bidirectional and splitting them to unidirectional ports did not raise problems, resulting in a clearer model. The evaluation of the OCL rules yielded the inconsistent ports and connectors (cf. Section 4.1 and Section 4.2) which were either removed or redefined.

[12]Automated Transfer Vehicle of the International Space Station, `http://www.esa.int/atv`

A second task was the verification of the uniqueness and completeness of ports, (cf. Section 4.3-Section 4.4). Approximately 20% of the evaluated ports were inconsistent with respect to rules 7 and 8. Figure 6 shows one such example.

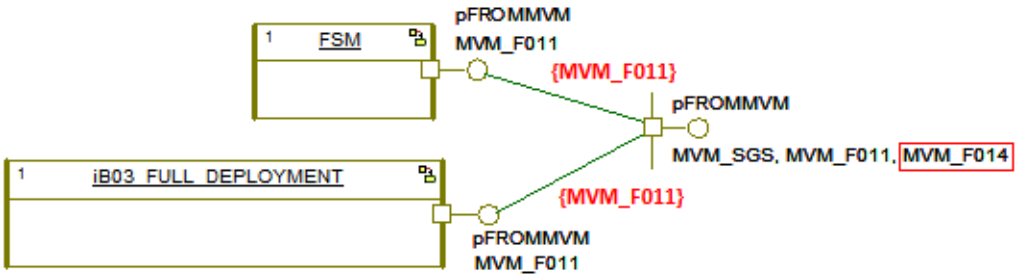

Figure 6: Inconsistent port with respect to uniqueness and completeness rules

Finally, the corrected model was given as input to the OMEGA2 compiler and was simulated with the IFx2 toolset. During simulation, deadlocks due to missing connectors or unhandled requests by ports were not found. Given the complexity of the models, this provides strong empirical evidence that, under the constraints of the rule set, the OMEGA2 type system is safe.

8 Conclusions

Composite structures play an important role in modelling real-time embedded systems. They offer a clear structure of these systems and an initialization scheme for the objects contained. They are a big evolution of the UML standard version 2.x, since in the version 1.4 the initialization order of complex systems was user-defined. Since the standard is ambiguous and semantic variation point left open, we proposed to define a rule set observing composite structure and to prove its type safety.

We presented a definition and formalization of an operational model of UML composite structures, our approach being based on :

- dynamic typing of connectors based on a derived notion of *transported interfaces*;
- a set of static well-formedness rules, including type checking rules;
- a full definition of the default behaviour of *Ports*, and the means for defining port behaviour differing from the default (by using implicit port associations, etc.)
- rules for relating composite structures with the concurrency model.

The rule set defined in Chapter 6 is used by the type checker of the OMEGA UML compiler. In addition, the compiler goes all the way down to an operational implementation of composite structures, by translating OMEGA UML models (edited with any XMI

2.0 compatible UML editor) into IF models, for which a simulation and model-checking platform exists allowing us to prove the correctness of UML embedded models.

Experiments have been conducted to prove that models observing this rule set are correct. While the OMEGA UML compiler is able to catch all modelling errors when translating the model into its IF description, the OCL formalisation can also reveal these issues in a step preceding the translation. Applying this formalisation on the model, it yields the elements that do not comply with our profile catching many corner cases.

The next step in our work is to prove the type-safety of our ruleset with respect to composite structures using the Isabelle/HOL proof assistant [16]. In this setting the type-safety means that: any request that travels through connectors reaches its terminus and every destination object receives only request compatible with its interfaces. Even thought we were able to show on realistic models using the simulation and exhaustive state-space search from IFx2 Toolset that no routing problems (deadlocks in ports due to missing links, unexpected requests not conforming to object interfaces, etc.) exist in the model, a formal proof is needed.

References

[1] ISO/IEC 8652/1995. *Ada 2005 Reference Manual. Language and Standard Libraries*, volume 4348 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM*, pages 237–267, 2004.

[3] Marius Bozga and Yassine Lakhnech. IF-2.0: Common Language Operational Semantics. Technical report, Verimag, 2002.

[4] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley, 2001.

[5] Arnaud Cuccuru, Sébastien Gérard, and Ansgar Radermacher. Meaningful Composite Structures. In *MoDELS*, pages 828–842, 2008.

[6] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Sci. Comput. Program.*, 55(1-3):81–115, 2005.

[7] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, Second International Symposium, FMCO 2003, Leiden, The Netherlands, November 4-7, 2003, Revised Lectures*, volume 3188 of *Lecture Notes in Computer Science*. Springer, 2004.

[8] Gregor Gößler and Joseph Sifakis. Priority systems. In de Boer et al. [7], pages 314–329.

[9] Object Management Group. Object Constraint Language, v2.2. Available at http://www.omg.org/spec/OCL/2.2/.

[10] Object Management Group. Systems Modeling Language, v1.1. Available at http://www.omg.org/spec/SysML/1.1/.

[11] Object Management Group. UML Profile for Modeling and Analysis of Real-Time Embedded Systems. Available at http://www.omgmarte.org.

[12] Object Management Group. Unified Modeling Language, v2.2. Available at http://www.omg.org/spec/UML/2.2.

[13] IBM. Rational Rhapsody v7.5. reference manuals. Available at http://www.ibm.com/developerworks/rational/.

[14] IFx Toolset. Available at http://www-omega.imag.fr/tools/IFx/IFx.php.

[15] ITU-T. Languages for telecommunications applications – Specification and Description Language (SDL). ITU-T Revised Recommendation Z.100, 1999.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2009.

[17] Iulian Ober and Iulia Dragomir. OMEGA2: A new version of the profile and the tools (regular paper). In *UML & AADL'2010 - 15th IEEE International Conference on Engineering of Complex Computer Systems, Oxford, Royaume Uni, 24/03/2010-25/03/2010*, pages 373–378, http://www.ieee.org/, 2010. IEEE.

[18] Iulian Ober, Susanne Graf, and Ileana Ober. A real-time profile for UML. *International Journal on Software Tools for Technology*, 8(2):113–127, 2006.

[19] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *International Journal on Software Tools for Technology*, 8(2):128–145, 2006.

[20] Ian Oliver and Vesa Luukkala. On UML's Composite Structure Diagram. In *5th Workshop on System Analysis and Modelling (SAM), Kaiserslautern, Germany*, June 2006.

[21] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley Professional Computing. John Wiley, 1994.

[22] TOPCASED, The Open-Source Toolkit for Critical Systems. Available at http://www.topcased.org/.

Appendix: OCL Formalization

```
-- HELPER FUNCTIONS

context Classifier

-- Verifies if the classifier is an interface
def: isInterface : Boolean = self.oclIsTypeOf(uml::Interface)

-- Verifies if the classifier is stereotyped with <<interfaceGroup>>
def: isInterfaceGroup : Boolean = self.getAppliedStereotypes()->select(
   name='interfaceGroup')->size()<>0

-- Verifies if the classifier is stereotyped with <<protected>>
def: isProtected : Boolean = self.getAppliedStereotypes()->select(name='
   protected')->size()<>0

--Verifies if the classifier is stereotyped with <<observer>>
def: isObserver : Boolean = self.getAppliedStereotypes()->select(name='
   observer')->size()<>0


context Port

-- Verifies if the port is reversed (the port requires interfaces)
-- def: isReversed : Boolean = self.getAppliedStereotypes()->select(name
   ='reversed')->size()<>0

-- Since the tool used for the development of our models is IBM Rhapsody
   tool we need to make some adjustments since Rhapsody supports the
   reversed mechanism and saves it as an attribute of RhpPort stereotype
def: isReversed : Boolean = self.getValue(self.getAppliedStereotypes()->
   select(name='RhpPort')->asOrderedSet()->at(1),'isReversed').oclAsType(
   Boolean)


context Connector

-- Verifies if the connector is of type port-port
def: has2Ports : Boolean = self.end.role->select(oclIsTypeOf(uml::Port))
   ->size() = 2

-- Verifies if the connector is of type part-part
def: has2Parts : Boolean = self.end.role->select(oclIsTypeOf(uml::
   Property) and not oclIsTypeOf(uml::Port))->size() = 2

-- Verifies if the connector is of type port-part
def: has1PartAnd1Port : Boolean = self.end.role->select(oclIsTypeOf(uml::
   Property) and not oclIsTypeOf(uml::Port))->size() = 1

-- For a port-port connector verifies if one of the ports is owned by an
```

```
   inner component and the other one by the composite structure
-- For a port-part connector verifies if the port is owned by an inner
   component
def: has1PartWithPort : Boolean = self.end.partWithPort -> reject(
   oclIsTypeOf(OclVoid))->size() = 1

-- For a port-port connector verifies if both ports are owned by inner
   components
def: has2PartsWithPort: Boolean = self.end.partWithPort -> reject(
   oclIsTypeOf(OclVoid))->size() = 2

-- For a port-port connector returns the first port from the set of ends
-- For a port-part connector returns the port
def: port1 : Port = self.end.role->select(oclIsTypeOf(uml::Port))->
   asOrderedSet()->at(1).oclAsType(uml::Port)

-- For a port-port connector returns the second port from the set of ends
def: port2 : Port = self.end.role->select(oclIsTypeOf(uml::Port))->
   asOrderedSet()->at(2).oclAsType(uml::Port)

-- For a part-part connector returns the first part from the set of ends
-- For a port-part connector returns the part
def: part1 : Property = self.end.role->select(oclIsTypeOf(uml::Property)
   and not oclIsTypeOf(uml::Port))->asOrderedSet()->at(1).oclAsType(uml::
   Property)

-- For a part-part connector returns the second part from the set of ends
def: part2 : Property = self.end.role->select(oclIsTypeOf(uml::Property)
   and not oclIsTypeOf(uml::Port))->asOrderedSet()->at(2).oclAsType(uml::
   Property)

-- Verifies if the connector is typed with an association
def: isTyped : Boolean = (not self.type.oclIsTypeOf(OclVoid))

-- For a part-part connector, in the case of a unidirectional association
    that may type the link we need to know which end is navigable such
   that we can determine the direction of the link
def: isEnd1Navigable : Boolean = self.end.definingEnd->asOrderedSet()->at
   (1).isNavigable()

-- Determines the ends of the association with which the link is typed
def: associationEnds : OrderedSet(Property) = self.type.memberEnd->
   asOrderedSet()

-- Verifies if both ends of the association with which a link is typed
   are navigable
def: isBidirectional : Boolean = associationEnds->select(isNavigable())->
   size() = 2

-- Verifies if an association with which a link is typed is not navigable
def: isNotNavigable : Boolean = associationEnds->select(isNavigable())->
```

```
    size() = 0

-- Verifies if the association is between two classes
def: isClassClassAssociation : Boolean = associationEnds->select(type.
    oclIsKindOf(uml::Class))->size()=2

-- Verifies if the association is between two interfaces
def: isInterfaceInterfaceAssociation : Boolean = associationEnds->select(
    type.oclIsKindOf(uml::Interface))->size()=2

-- For an association that types a link determines the origine
def: associationStartPoint : Property =
  if isBidirectional
    then self.associationEnds->at(1)
  else
    self.associationEnds->select(not isNavigable())->at(1)
  endif

-- For an unidirectional association that types a link determines origine
    's type
def: associationStartPointType : Type = associationStartPoint.type

-- For an unidirectional association that types a link determines the
    target
def: associationEndPoint : Property =
  if isBidirectional
    then self.associationEnds->at(2)
  else
    self.associationEnds->select(isNavigable())->at(1)
  endif

-- For an unidirectional association that types a link determines target'
    s type
def: associationEndPointType : Type = associationEndPoint.type

-- Verifies if the association is between a class and an interface and it
     has this direction
def: isClassInterfaceAssociation : Boolean = associationStartPointType.
    oclIsKindOf(uml::Class) and associationEndPointType.oclIsKindOf(uml::
    Interface)


-- RULE 1 AND RULE 2

context Connector

-- Definition of link's type
def: linkType : String =
  if has2Parts
    then 'assembly link between parts'
  else
```

```
    if has2Ports
      then if has1PartWithPort
             then if not port1.isReversed and not port2.isReversed
                    then 'inbound delegation link between provided ports'
                  else if port1.isReversed and port2.isReversed
                         then 'outbound delegation between required ports
                            '
                       else 'forbidden'
                       endif
                  endif
           else if (port1.isReversed and port2.isReversed) or
                   (not port1.isReversed and not port2.isReversed)
                  then 'forbidden'
                else  'assembly between provided-required ports'
                endif
           endif
    else
      if has1PartWithPort
        then if not port1.isReversed
               then 'assembly link between part and provided port'
             else 'assembly link between part and required port'
             endif
      else if not port1.isReversed
             then 'inbound delegation link between part and provided port
                '
           else 'outbound delegation link between part and required port'
           endif
      endif
    endif
  endif

-- Rule 1 and Rule 2
inv LinkType: self.linkType <> 'forbidden'


-- HELPER FUNCTIONS

context Connector

-- Verifies if a connector starts from the provided port given as
   parameter
def: isStartingFromProvidedPort (p:Port) : Boolean =
  (self.linkType = 'inbound delegation link between provided ports' and p
     .owner=self.owner) or
   self.linkType = 'inbound delegation link between part and provided
      port'

-- Verifies if a connector starts from the required port given as
   parameter
def: isStartingFromReversedPort (p:Port) : Boolean =
  (self.linkType = 'outbound delegation between required ports' and p.
```

```
      owner<>self.owner) or
    self.linkType = 'assembly between provided-required ports ' or
    self.linkType = 'assembly link between part and required port'

-- Verifies if a connector starts from the port given as parameter
def: isStartingFromPort (p:Port) : Boolean =
  isStartingFromProvidedPort(p) or (isStartingFromReversedPort(p) and p.
     isReversed)

-- Verifies if a connector starts from a part
def: isStartingFromPart : Boolean =
  self.linkType = 'assembly link between part and provided port' or
  self.linkType = 'outbound delegation link between part and required
     port' or
  self.linkType = 'assembly link between parts'

-- For a port-port connector determines the port from which the link
    starts
-- For a port-part connector the starting port is the only port of the
    link
def: linkStartPort : Port =
  if has2Ports
    then
      if isStartingFromPort(port1)
        then port1
      else port2
      endif
  else port1
  endif

-- For a port-port connector determines the port in which the link ends
-- For a port-part connector the ending port is the only port of the link
def: linkEndPort : Port =
  if has2Ports
    then if isStartingFromPort(port1)
           then port2
         else port1
         endif
  else port1
  endif

-- For a port-part connector the starting part is the only part of the
    link
-- For a part-part connector determines the part from which the link
    starts (this part it is not navigable)
def: linkStartPart : Property =
  if has1PartAnd1Port
    then part1
  else
    if isEnd1Navigable
      then part2
```

```
    else part1
    endif
  endif

-- For a port-part connector the ending part is the only part of the link
-- For a part-part connector determines the part in which the link ends (
   this part it is navigable)
def: linkEndPart : Property =
  if has1PartAnd1Port
    then part1
  else
    if isEnd1Navigable
      then part1
    else part2
    endif
  endif


-- PROVIDED / REALIZED INTERFACES

context Port

-- Definition of interfaces realized by a port
def: interfaces : Set(Classifier) = self.provided->reject(
   isInterfaceGroup)

context Classifier

-- Definition of classifier's parents recursive computation
def: getParentsRec : Set(Classifier) =  self.general->union(self.general
   ->iterate(p:Classifier; res:Set(Classifier)=Set{}| res->union(p.
   getParentsRec)))

context Class

-- Definition of all interfaces directly realized by a class
def: iRealizations : Set(Classifier) = self.interfaceRealization.contract
   ->asSet()

-- Definition of interfaces provided by a class directly or indirectly
   realized (used in the case of a link not typed by an association)
def: interfaces : Set(Classifier) =
  iRealizations->union(iRealizations->iterate(i:Interface; res:Set(
     Classifier)=Set{}| res->union(i.getParentsRec)))
  ->union(self.getParentsRec->iterate(c:Class; res:Set(Classifier)=Set{}|
      res->union(c.interfaces)))
  ->reject(isInterfaceGroup)

context Interface

-- Definition of interfaces provided by an interface (used in the case of
```

```
    a link typed by an association pointing to an interface)
def: interfaces : Set(Classifier) = self.oclAsType(uml::Classifier)->
   asSet()->union(self.getParentsRec)->select(not isInterfaceGroup and
   isInterface)

context Type

-- Determines the set of provided interfaces by a class or an interface
def: interfaces : Set(Classifier) =
  if self.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces
  else
    self.oclAsType(uml::Class).interfaces
  endif


-- RULE 6

context Connector

-- Definition of the set of transported interfaces
def: setTransportedInterfaces : Set(Classifier) =
  if has2Parts
    then Set{OclInvalid}
  else if has2Ports
         then if isTyped
                then if isStartingFromPort(port1)
                       then (port1.interfaces) -> intersection(
                          associationEndPointType.interfaces)
                     else (port2.interfaces) -> intersection(
                        associationEndPointType.interfaces)
                     endif
              else (port1.interfaces) -> intersection(port2.interfaces)
              endif
       else
         if isTyped
           then (port1.interfaces) -> intersection(
              associationEndPointType.interfaces)
         else (port1.interfaces) -> intersection(part1.type.interfaces)
         endif
       endif
  endif

-- Rule 6
inv SetOfTransportedInterfacesNonEmpty: self.setTransportedInterfaces->
   size()<>0


-- HELPER FUNCTIONS

context Classifier
```

```
-- Verifies if the current classifier is compatible with the one given as
def: isCompatible(c:Classifier) : Boolean =
  if self.oclIsKindOf(uml::Interface) and c.oclIsKindOf(uml::Interface)
    then self.oclAsType(uml::Interface).interfaces->includes(c)
  else
    if self.oclIsKindOf(uml::Class) and c.oclIsKindOf(uml::Interface)
      then self.oclAsType(uml::Class).interfaces->includes(c)
    else
     (c->asSet()->union(c.getParentsRec))->includes(self)
    endif
  endif

context Type

-- Verifies if the link end's type (the type of a part) is compatible
   with the association end's type given as parameter
def: isCompatible(t:Type) : Boolean = self.oclAsType(uml::Classifier).
   isCompatible(t.oclAsType(uml::Classifier))

context Port

-- Verifies if the current port is compatible with the association end's
   type given as parameter
def: isCompatible(t:Type) : Boolean = self.interfaces->includesAll(t.
   oclAsType(uml::Interface).interfaces)


-- RULE 3 AND RULE 4

context Connector

-- Verifies if a link starting from a port and typed with an association
   has the same direction with the association and the interface pointed
   is included in the set of transported interfaces
def: linkStartingFromPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isInterfaceInterfaceAssociation
        then if has1PartAnd1Port
              then linkStartPort.isCompatible(associationStartPointType)
                  and linkEndPart.type.isCompatible(
                  associationEndPointType) and
                    setTransportedInterfaces->includes(
                        associationEndPointType.oclAsType(uml::Classifier))
             else linkStartPort.isCompatible(associationStartPointType)
                 and linkEndPort.isCompatible(associationEndPointType) and
                  setTransportedInterfaces->includes(
                     associationEndPointType.oclAsType(uml::Classifier))
             endif
```

```
    else false
    endif
  endif

-- Verifies if a link starting from a port is well-formed
def: linkStartingFromPort : Boolean =
  if (not isStartingFromPart) and isTyped
    then linkStartingFromPortVerification
  else true
  endif

-- Rule 3 and Rule 4
inv LinkStartingFromPort: self.linkStartingFromPort


-- RULE 3 AND RULE 5

context Connector

-- For a link between two parts verifies if the ends are compatible with
   the corresponding ends of the accepted association
def: linkPartPartVerification : Boolean =
  if isNotNavigable
    then false
  else
    if isBidirectional
      then (linkStartPart.type.isCompatible(associationStartPointType)
          and linkEndPart.type.isCompatible(associationEndPointType)) or
            (linkStartPart.type.isCompatible(associationEndPointType) and
               linkEndPart.type.isCompatible(associationStartPointType))
    else (linkStartPart.type.isCompatible(associationStartPointType) and
       linkEndPart.type.isCompatible(associationEndPointType))
    endif
  endif

-- For a link between a part and a port verifies if the ends are
   compatible with the corresponding ends of accepted association
def: linkPartPortVerification : Boolean =
  if isNotNavigable or isBidirectional
    then false
  else
    if isClassClassAssociation
      then false
    else
      if isInterfaceInterfaceAssociation
        then (linkStartPart.type.isCompatible(associationStartPointType)
           and linkEndPort.isCompatible(associationEndPointType))
      else
        if isClassInterfaceAssociation
          then linkStartPart.type.isCompatible(associationStartPointType)
              and linkEndPort.isCompatible(associationEndPointType)
```

```
        else false
        endif
      endif
    endif
  endif

-- Verifies if a link starting from part is typed with an association and
     if it is well-formed
def: linkStartingFromPart : Boolean =
  if isStartingFromPart
    then if has2Parts
           then isTyped and linkPartPartVerification
         else isTyped and linkPartPortVerification
         endif
  else true
  endif

-- Rule 3 and Rule 5
inv LinkStartingFromPart: self.linkStartingFromPart


-- HELPER FUNCTIONS

context Port

-- Determines all the connectors starting or ending in a port
def: connectors : Set(Connector) = self.end.owner.oclAsType(uml::
    Connector)->asSet()

-- Verifies if a port has connectors starting or ending in it
def: hasConnectors : Boolean = self.connectors->size()>1

-- Determines all the connectors starting or ending in a port that are
    not typed with an association
def: connectorsNotTyped : Set(Connector) = self.connectors->select(type.
    oclIsTypeOf(OclVoid))

-- Verifies if a port has connectors not typed with an association that
    are starting or ending in it
def: hasConnectorsNotTyped : Boolean = self.connectorsNotTyped->size()>1

-- Determines all the connectors starting from a port (if the boolean
    parameter is true it collects all the connectors; if it is false it
    collects only the connectors not typed with association)
def: connectorsStartingFromPort(withType:Boolean) : Set(Connector) =
  if withType then
        self.connectors->select(c:Connector|c.isStartingFromPort(self))
  else
        self.connectorsNotTyped->select(c:Connector|c.isStartingFromPort(
           self))
  endif
```

```
-- Verifies if a port is the origine for at least one connector
def: isStartingPort : Boolean = self.connectorsStartingFromPort(true)->
    size() >= 1

-- Determines the union of the sets of transported interfaces on each
    link starting from the port
def: unionSetForTransportedInterfacesOnLinks(withType:Boolean) : Set(
    Classifier) =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; s:Set(
      Classifier)=Set{} | s->union(c.setTransportedInterfaces))

-- Determines the sum of the number of transported interfaces on each
    link starting from the port
def: noOfTransportedInterfacesOnLinks(withType:Boolean) : Integer =
  self.connectorsStartingFromPort(withType)->iterate(c:Connector; i:
      Integer=0 | i + (c.setTransportedInterfaces->size()))


-- RULE 7

context Port

-- Definition of pairwise disjoint sets of transported interfaces
context Port
def: isPairwiseDisjoint : Boolean =
  if self.connectorsStartingFromPort(false)->size() >= 2
    then  self.unionSetForTransportedInterfacesOnLinks(false)->size() =
        self.noOfTransportedInterfacesOnLinks(false)
  else true
  endif

-- Rule 7
inv PairwiseDisjoint: self.isPairwiseDisjoint


-- RULE 8

context Port

-- Verifies if the union of sets of transported interfaces is equal to
    the interfaces provided/required
def: isComplete : Boolean =
  if self.isStartingPort
    then unionSetForTransportedInterfacesOnLinks(true) = self.interfaces
  else true
  endif

-- Rule 8
inv Completeness: self.isComplete
```

```
-- HELPER FUNCTIONS

context Class

-- Determines the number of parts of a composite structure
def: noOfComponents : Integer = self.part->size()

-- Verifies if a class is a composite structure
def: isComposite : Boolean = noOfComponents <> 0

-- Determines the number of active parts owned by a composite structure
def: noOfActiveComponents : Integer = self.part.type.oclAsType(uml::Class
   )->select(isActive=true)->size()

-- Determines the number of passive parts owned by a composite structure
def: noOfPassiveComponents : Integer = self.part.type.oclAsType(uml::
   Class)->select(isActive=false and not isProtected)->size()

-- Determines the number of protected parts owned by a composite
   structure
def: noOfProtectedComponents : Integer = self.part.type.oclAsType(uml::
   Class)->select(isProtected)->size()


-- RULE 9 AND RULE 10

context Class

-- Definition of a well-formed class
def: isWellFormed : Boolean =
  if self.isActive and isComposite
    then  if noOfActiveComponents + noOfProtectedComponents =
       noOfComponents or
            noOfPassiveComponents = noOfComponents
           then true
         else false
         endif
  else
    if (not self.isActive) and (not isProtected) and isComposite
      then if noOfPassiveComponents = noOfComponents
             then true
           else false
           endif
    else OclInvalid
    endif
  endif

-- Rule 9 and Rule 10
inv CompositeStructure: self.isWellFormed <> false
```

```
-- RULE 11

context Class

-- Determines the number of observer parts owned by a composite structure
def: noOfObservers : Integer = self.part.type.oclAsType(uml::Class)->
    select(isObserver)->size()

-- Definition of a well formed observer
def: isObserverWellFormed : Boolean =
  if self.isObserver and isComposite
    then noOfComponents = noOfObservers
  else true
  endif

-- Rule 11
inv CompositeObserver: self.isObserverWellFormed <> false
```